Advanced Java Week 3 - Day 03

Agenda:

1) Functional Programming
2) Object Oriented Programming vs Functional Programming
3) Features of Functional Programming
4) Functional Programming Techniques
5) Functional Interface
6) Lambda Expression
7) Functional Interface and its Implementations Examples
8) Some In-Built Functional Interfaces

Functional Programming:

Functional programming is a programming paradigm — a style of building the structure and elements of computer programs — that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data — Wikipedia

Functional programming is a programming paradigm which concentrates on computing results rather than on performing actions. i.e., when you call a function, the only significant effect that the function has is usually to compute a value and return it.

Object Oriented Programming vs Functional Programming

Object Oriented programming deals with Objects, whereas functional programming deals with functions.

Object Oriented Programming implements imperative programming which focuses on how a particular function has to be carried out, whereas Imperative programming focuses on what to be done rather than how to be done, thus functional programming drastically reduces the code size
A programming language is said to have first-class functions if it treats functions as first-class citizens. This means that functions are allowed to support all operations typically available to other entities. These include assigning functions to variables, passing them as arguments to other functions and returning them as values from other functions.

This property makes it possible to define higher-order functions in functional programming. Higher-order functions are capable of receiving functions as arguments and returning a function as a result. This further enables several techniques in functional programming such as function composition and currying.
Traditionally, it was only possible to pass functions in Java using constructs such as functional interfaces or anonymous inner classes. Functional interfaces have exactly one abstract method and are also known as Single Abstract Method (SAM) interfaces.

```
Collections.sort(numbers, new Comparator <Integer>(){
        @Override
        public int compare(Integer n1,Integer n2)
        {
                Return n1.compareTo(n2);
        }
}
```

As we can see, this is a tedious and verbose technique - certainly not something that encourages developers to adopt functional programming.
Fortunately , Java 8 brought many new features to ease the process, such as Lambda Expressions , Method references and predefined functional interfaces.

Lambda Expressions can help us with the same task as follows
Collections.sort(numbers, (n1, n2) -> n1.compareTo(n2));
This is more concise and understandable.

However please note that while this may give us the impression of using functions as first-class citizens in Java, thats; not the case.
Behind the syntactic sugar of lambda expressions. Java still wraps these into Functional interfaces, So java treats lambda expressions as on Object. Which is the true first-class citizen in Java.

The features of Functional Programming are:

1) Pure Functions
2) Immutability
3) Referential Transparency
4) Recursion
5) Higher Order Functions

PURE FUNCTION:

The definition of pure function emphasizes that a pure function should return a value based only on arguments and should have no side effects.
This can sound quite against some of the best practices in Java
As an Object Oriented language, Java recommends encapsulation as a core programming practice. It encourages hiding an object's internal state and exposing only essential methods to access and modify it.
So, these methods are not strictly pure functions

Of course encapsulation and other Object - Oriented principles are only recommendations and not binding in java. In fact , developers have recently started to realize the value of defining immutable states and methods without side effects
Let's say we want to find the sum of all the numbers we have just sorted:

```
Integer sum(List <Integer> numbers) {
        Return numbers.stream().collect(Collectors.summingInt(Integer::intValue));
}
```

Refer: "streams" summary for more info on streams

This method depends only on the arguments it receives, so it is deterministic. Moreover, it does not lead to any sort of side effects.
Side effects can be anything apart from the intended behavior of the method. For instance, side effects can be as simple as updating a local or global state or saving to a database before returning a value (Purists also treat logging as a side effect)
Benefits of Functional Programming :

1) Maintainable
2) Concise
3) Focus on Problem rather than Code
4) Facilitates Parallelism
5) Independent of Testing modules

IMMUTABILITY:

Immutability is one of the core principles of functional Programming and it refers to the property that an entity can't be altered or changed after being instantiated.
In a functional programming language. This is supported by design at the language level. But in Java we have to make our own decision to create immutable data structures.
Java itself provides several pre-defined immutable types, for instance String. This is primarily for security reasons because we widely use String in class loading and as keys in hash-based data structures.
There are also several other pre-defined immutable types such as primitive wrappers and math types.
But what about the data structures we create in Java? they are not immutable by default , and we have to make a few changes to attain immutability.
The use of the final keyword is one of them, but it does not stop there

```
public class ImmutableData {
    private final String someData;
    private final AnotherImmutableData anotherImmutableData;
    public ImmutableData(final String someData, final AnotherImmutableData anotherImmutableData) {
        this.someData = someData;
        this.anotherImmutableData = anotherImmutableData;
    }
    public String getSomeData() {
        return someData;
    }
    public AnotherImmutableData getAnotherImmutableData() {
        return anotherImmutableData;
    }
}

public class AnotherImmutableData {
    private final Integer someOtherData;
    public AnotherImmutableData(final Integer someData) {
        this.someOtherData = someData;
    }
    public Integer getSomeOtherData() {
        return someOtherData;
    }
}
```

Note that we have to sincerely follow a few rules:

1) All fields of an immutable data structure must be immutable.
2) This is applicable to all the nested types and collections (including what they contain) as well.
3) There should be one or more constructors for initialization as needed.
4) There should only be accessor/getter methods, possibly with no side effects.

It's not easy to get this right completely every time, especially when the data structures start to get complex.
However several external libraries can make working with immutable data in java simpler, For instance , Immutables and Project Lombok provide ready to use frameworks for defining immutable data structures in Java.

Referential Transparency

Referential Transparency is probably one of the more difficult principles of functional programming to understand, but the concept is actually simple to understand.

We call an expression referentially transparent if replacing it with its corresponding value has no negative effect on the program's behavior.
This enables some powerful techniques in functional programming such as higher-order functions and lazy/late evaluation.
To understand this better let's look at the following example.

```
public class SimpleData {
    private Logger logger = Logger.getGlobal();
    private String data;
    public String getData() {
        logger.log(Level.INFO, "Get data called for SimpleData");
        return data;
    }
    public SimpleData setData(String data) {
        logger.log(Level.INFO, "Set data called for SimpleData");
        this.data = data;
        return this;
    }
}
```

This is a typical POJO class in java , but we’re interested in finding if this provides referential transparency
Let’s observe the following statements

```
String data = new SimpleData().setData("Baeldung").getData();
logger.log(Level.INFO, new SimpleData().setData("Baeldung").getData());
logger.log(Level.INFO, data);
logger.log(Level.INFO, "Baeldung");
```

The three calls to logger are logically equivalent but not referentially transparent.
The first call is not referentially transparent since it produces a side effect. If we replace this call with its value as in the third call, we’ll miss the logs.

The second call is also not referentially transparent since we can alter SimpleData so it is mutable, A call to data.setData anywhere in the program would make it difficult for it to be replaced with its value.

So for referential Transparency, we need our functions to be pure and immutable. These two pre conditions which are discussed above.

As an interesting outcome of Referential Transparency, we produce context-free code. In other words we can run them in any order and context, which leads to different optimization possibilities.

Recursion:

Recursion is another powerful technique in functional programming that allows one to break down a problem into smaller pieces of problems. The main advantage of recursion is that it helps us in eliminating the side effects, which is typical of any imperative style looping.

Let us take an example of calculating the factorial using recursion

```
Integer factorial(Integer number) {
   return (number == 1) ? 1 : number * factorial(number - 1);
}
```

Here above we are calling the same function recursively until we reach the base case and then start to compute our required result.

We are making the recursive call before calculating the result at each step or in other words at the head of the computation, so this style of recursion is called as “Head Recursion”, A disadvantage of this method is that every step has to hold the state of all previous steps until we achieve the base case. This is not really a problem for small numbers , but holding the state for large numbers can be inefficient.

A solution is slightly different implementation of recursion called “Tail Recursion”

```
Integer factorial(Integer number, Integer result) {
   return (number == 1) ? result : factorial(number - 1, result * number);
}
```

Please note in the latter example an accumulator in the function, eliminating the need to hold the state at every step of recursion. The real benefit of this style is to make use of the compiler optimizations, where the compiler can decide to release the current function’s stack frame, a technique also known as tail-call elimination.

Higher Order Functions:

A higher order function is a function that either takes a function (method) as parameter, or returns a function after its execution.

```
List<String> list = new ArrayList<>();
list.add("One");
list.add("Abc");
list.add("BCD");

Collections.sort(list, (String a, String b) -> {
   return a.compareTo(b);
});

System.out.println(list);
```

The Collection.sort() takes two parameters. The first parameter is a List and the second parameter is a lambda (function). The lambda parameter is what makes Collections.sort() a higher order function.

Functional Programming Techniques

The Functional Programming principles that are discussed earlier enable us to use several techniques to benefit from functional programming.

Function Composition:

Refers to composing complex functions by combining simpler functions.

This is primarily achieved in Java using functional interfaces, which are target types for lambda expressions and method references.

Typically any interface with only a single abstract method can behave as a functional interface. So we can define a functional interface quite easily.

However Java 8 provides us many functional interfaces by default for different use cases under the package java.util.function.

Many of these functional interfaces provide support for function composition in terms of default and static methods.
To understand Functional Interfaces better.
Function is a simple and generic functional interface that accepts one argument and produces a result.
It also provides two default methods, compose and andThen, which will help us in function composition

```
Function<Double, Double> log = (value) -> Math.log(value);
Function<Double, Double> sqrt = (value) -> Math.sqrt(value);
Function<Double, Double> logThenSqrt = sqrt.compose(log);
logger.log(Level.INFO, String.valueOf(logThenSqrt.apply(3.14)));
// Output: 1.06
Function<Double, Double> sqrtThenLog = sqrt.andThen(log);
logger.log(Level.INFO, String.valueOf(sqrtThenLog.apply(3.14)));
// Output: 0.57
```

Both of these methods allow us to compose multiple functions into a single function but offer different semantics. While the *compose* method applies the function passed in the argument first and then the function on which it is invoked.
andThen does the same in reverse.

Several other functional interfaces have interesting methods to use in function composition, such as the default methods and, or and negate in the Predicate interface. While these functional interfaces accept a single argument, There are two-arity specializations, such as *BiFunction* and *BiPredicate*.

Monads:
Many of the functional programming concepts derive from Category Theory, which is a general theory of functions in mathematics. It presents several concepts of categories such as functors and natural transformations.

Formally a monad is an abstraction that allows structuring programs generically. So a monad allows us to wrap a value, apply a set of transformations and get the value back with all transformations applied.

There are 3 laws that any monad needs to follow - left identity, right identity and associativity

There are few monads in Java such as Optional and Stream

```
Optional.of(2).flatMap(f -> Optional.of(3).flatMap(s -> Optional.of(f + s)))
```

Why is “optional” called a Monad?

Refer: “optional” summary for more info on “optionals”
Here “Optional” allows us to wrap a value using the method of and apply a series of transformations. We are applying the transformation of adding another wrapped value using the method flatMap.
We could show that Optional follows the three laws of monads. However , an Optional does break the monad laws under some circumstances. But it should be good enough for us for most practical situations.

If we understand monads’ basics , we will soon understand that there are many other examples in Java such as Stream and CompletableFuture , They help us attain different objectives. But they all have a standard composition in which context manipulation or transformation is handled.

CURRYING :

Currying is a mathematical technique of converting a function that takes multiple arguments into a sequence of functions that take a single argument.
In functional Programming, it gives us a very powerful composition technique where we don't need to call a function with all its arguments.
Moreover , a curried function does not realize its effect until it receives all the arguments.

In pure functional programming languages such as Haskell, currying is well supported .In fact all functions are curried by default.

In the following example

```
Function<Double, Function<Double, Double>> weight = mass -> gravity -> mass * gravity;

Function<Double, Double> weightOnEarth = weight.apply(9.81);
logger.log(Level.INFO, "My weight on Earth: " + weightOnEarth.apply(60.0));

Function<Double, Double> weightOnMars = weight.apply(3.75);
logger.log(Level.INFO, "My weight on Mars: " + weightOnMars.apply(60.0));
```

We have defined a function to compute the weight of a planet. While the mass remains same, gravity varies by the planet.

We can partially apply the function by passing the gravity alone to define a function for a particular planet. Moreover we can also pass this partially applied function around as an argument or return value for arbitrary composition.

Currying depends on the language to provide 2 basic features : Lambda Expression & closures.
“Closures” is not in the scope of this document.

Functional Interface :

- An interface that permits only one abstract method inside them
- Also called as SAM(Single Abstraction Method) Interfaces
- Can have default or static methods
- If an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface's abstract method count
- @FunctionalInterface annotation is used to ensure that the functional interface can't have more than one abstract method

Lambda Expressions:

Lambda expressions provide a clear and concise way to represent one method interface using an expression

- They are similar to anonymous implementations of single method interfaces, but with a more concise syntax.
- They can be passed around as if it is an object and executed on demand
- They can see and use nearby values, but can't change them

Functional Interface and its Implementation Examples:

Example 1:

```
interface MyInterface1{
 public void display();
}
```

Functional Interface implementation Through Lambda Expression:

```
        MyInterface1 inter1 = () -> {

 System.out.println("Welcome to Functional Interfaces");
 System.out.println("And Implementations Through Lambda Expressions ");
 };
```

Functional Interface Method Invocation:

```
inter1.display();
```

—--------------------

Example 2:

Functional Interface:

```
interface MyInterface2{
  public void calculate(int a, int b);
}
```

Implemented Through Lambda Expressions:

```
MyInterface2 inter21 = (int a, int b) -> {
   int result = a + b;
  System.out.println("The sum is "+result);
  };

MyInterface2 inter22 = (int a, int b) ->{
   int result = a * b;
   System.out.println("The Product is "+result);
  };
```

Method Invocations:

```
  inter21.calculate(100, 20);
  inter22.calculate(100,20);
```

Example 3:

Functional Interface

```
interface Customer
{
public void calculateInvoiceAmt(int qty,int price);
}
```

Implementation through Lambda Expression:

```
Customer customer1 = (int qty, int price) -> {
int invoiceAmt = qty * price;
if(invoiceAmt >= 10000)
{
System.out.println(" Invoice Amt is "+invoiceAmt+" Good Purchase..");
}
else
{
System.out.println(" Invoice Amt is "+invoiceAmt+" Moderate Purchase..");
}
  };
```

Invocation :

```
  customer1.calculateInvoiceAmt(100, 120);
  customer1.calculateInvoiceAmt(100, 50);
```

Example 4:

Functional Interface

```
interface NewCustomer
{
  public double calculateInvoice(int qty,int price);
}
interface SalesData
{
  public void calculateInvoice(int qty,int price,NewCustomer nc);
}
```

Lambda Expression

```
 NewCustomer nCustomer1 = (int qty, int price) -> {


   double invAmt = qty * price;
   double finalInvAmt = invAmt - (0.1 * invAmt);
   return finalInvAmt;
   };


 NewCustomer nCustomer2 = (int qty,int price) ->{
   double invAmt = qty * price;
   double finalInvAmt = invAmt - (0.2 * invAmt);
   return finalInvAmt;
   };
```

Invocations:

```
 double customer1InvAmt = nCustomer1.calculateInvoice(1000, 100);
   System.out.println("The Invoice Amount for First Customer "+customer1InvAmt);



 double customer2InvAmt = nCustomer2.calculateInvoice(1000, 100);
   System.out.println("The Invoice Amount for Second Customer "+customer2InvAmt);
```

Lambda Expression:

```
 SalesData sd1 = (int qty , int price, NewCustomer nc) -> {
  double actInvoiceValue = qty * price;
  System.out.println("The Actual Invoice Value is "+actInvoiceValue);
  if(actInvoiceValue >= 100000)
  {
  System.out.println("Prime Customer");
  }
  else
  {
  System.out.println("Regular Customer ");
  }
double discountedInvValue = nc.calculateInvoice(qty, price);
System.out.println("The Discounted Invoice Value is "+discountedInvValue);
  };
System.out.println("-------------------------------");
  System.out.println("First Customer Details....");
  sd1.calculateInvoice(1000, 120, nCustomer1);
  System.out.println("Second Customer Details....");
  sd1.calculateInvoice(1000, 120, nCustomer2);
```

Built In Functional Interfaces:

Supplier<T>	A Supplier<T> is used to generate objects/results when no input/parameter is required. Has **get()** method.
Predicate<T>	A Predicate<T> is used to hold/encapsulate a condition based on a given parameter that could be used later in code specially in other methods that takes Predicate as parameter. Has **test()** method.
BiPredicate<T, U>	BiPredicate<T, U> is similar to Predicate but here you can build condition based on 2 given parameters. Has **test()** method.
Consumer<T>	Consumer<T> is used to do something with a parameter but not return anything. Has **accept()** method.
BiConsumer<T, U>	BiConsumer<T, U> is similar to Consumer<T> but it accepts 2 parameters to do something. Has **accept()** method.
Function<T, R>	Function<T, R> is used to do exactly what you can do with a java method but it takes only one parameter to do something and returns a value. Has **apply()** method.
BiFunction<T, U, R>	BiFunction<T, U, R> same as Function<T, R> but it can take 2 parameters to do something. Has **apply()** method.

CONSUMER :

- A Consumer is a functional interface that accepts a single input and returns no output. In layman's language, as the name suggests the implementation of this interface consumes the input supplied to it.
- Consumer interface has two methods:
- void accept(T t);

default Consumer<T> andThen(Consumer<? super T> after);
Sample:

```
 public void whenNamesPresentConsumeAll(){
   Consumer<String> printConsumer = t -> System.out.println(t);
   Stream<String> cities = Stream.of("Chennai", "Goa", "Bangalore", "Mumbai");
   cities.forEach(printConsumer);
 }
public void whenNamesPresentUseBothConsumer(){
    List<String> cities = Arrays.asList("Chennai", "Goa", "Bangalore", "Mumbai");
    Consumer<List<String>> upperCaseConsumer = list -> {
        for(int i=0; i< list.size(); i++){
            list.set(i, list.get(i).toUpperCase());
        }
    };
    Consumer<List<String>> printConsumer = list -> list.stream().forEach(System.out::println);
    upperCaseConsumer.andThen(printConsumer).accept(cities);
}
```

The above sample displays Cities in Upper Case

SUPPLIER:

- A Supplier is a simple interface which indicates that this implementation is a supplier of results. This interface, however, does not enforce any restrictions that supplier implementation needs to return a different result on each invocation.
- The supplier has only one method get() and does not have any other default and static methods.

Sample:

```
public void supplier(){
Supplier<Double> doubleSupplier1 = () -> Math.random();
DoubleSupplier doubleSupplier2 = Math::random;

System.out.println(doubleSupplier1.get());
System.out.println(doubleSupplier2.getAsDouble());
}
```

- The supplier interface has its primitive variant

```
IntSupplier int getAsInt();
DoubleSupplier double getAsDouble();
LongSupplier long getAsLong();
BooleanSupplier boolean getAsBoolean();
```

_

```
public void supplierWithOptional(){
Supplier<Double> doubleSupplier = () -> Math.random();
Optional<Double> optionalDouble = Optional.empty();
System.out.println(optionalDouble.orElseGet(doubleSupplier));
}
```

—---EOD—--